Applications of Stack Data Structure

Expression Representations: Infix, Postfix, Prefix

For a binary operator, if the operator is written in between the two operands upon which it operates, then this representation is known as infix representation. Example: $a + b$.

In the above example, a and b are the operands, and + is the operator. Since we have written the expression with the operator in between the two operands, this is an example of infix representation.

For a binary operator, if the operator is written after the two operands upon which it operates, then this representation is known as postfix representation. Example: $a\ b\ +$.

In the above example, a and b are the operands and + is the operator. Since we have written the expression with the operator after the two operands, this is an example of the postfix representation.

Similarly, we also have the prefix representation, in which the operator is written before the operands (e.g., $+\ a\ b$).

Infix to Postfix Conversion

While people are more comfortable with infix representation, there are certain advantages to computationally process expressions in their postfix form. Thus, it is useful to be able to convert from one form to another.

The following algorithm specifies how an expression in the infix notation may be converted to an equivalent expression in the postfix notation with the help of a stack data structure.

Algorithm InfixToPostfix
Input: A string containing a valid expression in infix notation.
Output: A string containing the equivalent expression in postfix notation.

```
1. For each symbol in the infix string do
2.     If the symbol is an operand, append it to the postfix string.
3.     If the symbol is a left parenthesis then push it onto the stack.
4.     If the symbol is a right parenthesis then pop symbols (operators) from the stack and append them to the postfix string until a left parenthesis is popped. Discard the parentheses.
5.a    If the symbol is a left associative operator then pop and append to the postfix string every operator on the stack that has precedence higher than or equal to the current operator. Next push the current operator onto the stack.
5.b    If the symbol is a right associative operator then pop and append to the postfix string every operator on the stack that has precedence higher than the current operator. Next push the current operator onto the stack.
6. When the end of the infix string is observed, pop all remaining operators and append to the postfix string.
```

Example 1

Convert the expression (A + B * C / (D – E)) / F from infix to postfix.

Next Symbol	**Postfix String**	**Stack**
(		(
A	A	(
+	A	(+
B	A B	(+
*	A B	(+ *
C	A B C	(+ *
/	A B C *	(+ /
(	A B C *	(+ / (
D	A B C * D	(+ / (
-	A B C * D	(+ / (-
E	A B C * D E	(+ / (-
)	A B C * D E -	(+ /
)	A B C * D E - / +	
/	A B C * D E - / +	/
F	A B C * D E - / + F	/
	A B C * D E - / + F /	

Example 2

Convert the expression (2 + 3 ^ 2 ^ 3 - 6 * (15 - 3) + 4) from infix to postfix.

Next Symbol	Postfix String	Stack
(		(
2	2	(
+	2	(+
3	2 3	(+
^	2 3	(+ ^
2	2 3 2	(+ ^
^	2 3 2	(+ ^ ^
3	2 3 2 3	(+ ^ ^
-	2 3 2 3 ^ ^ +	(-
6	2 3 2 3 ^ ^ + 6	(-
*	2 3 2 3 ^ ^ + 6	(- *
(	2 3 2 3 ^ ^ + 6	(- * (
15	2 3 2 3 ^ ^ + 6 15	(- * (
-	2 3 2 3 ^ ^ + 6 15	(- * (-
3	2 3 2 3 ^ ^ + 6 15 3	(- * (-
)	2 3 2 3 ^ ^ + 6 15 3 -	(- *
+	2 3 2 3 ^ ^ + 6 15 3 - * -	(+
4	2 3 2 3 ^ ^ + 6 15 3 - * - 4	(+
)	2 3 2 3 ^ ^ + 6 15 3 - * - 4 +	
	2 3 2 3 ^ ^ + 6 15 3 - * - 4 +	

Postfix Expression Evaluation

A benefit of the postfix representation is that it can be evaluated efficiently. The following algorithm shows how we need to perform only a single left-to-right scan of an expression in postfix notation in order to evaluate it. Again, we make use of the stack data structure for performing this operation.

Algorithm PostfixEvaluation
Input: A string containing an expression in postfix notation.
Output: The resultant value of the expression.

```
1. For each symbol in the infix string do
2.     If the symbol is an operand, push it onto the stack.
3.     If the symbol is an operator do
4.         Pop the stack and assign the operand to variable right
5.         Pop the stack and assign the operand to variable left
6.         Evaluate left operator right
7.         Push the result back onto the stack
8. Pop the stack to obtain the final result
```

Example 1

Evaluate the following postfix expression: 5 6 5 * 18 15 - / + 2 /

Next Symbol	**Stack**
5	5
6	5 6
5	5 6 5
*	5 30
18	5 30 18
15	5 30 18 15
-	5 30 3
/	5 10
+	15
2	15 2
/	7.5

Answer: 7.5

Verify that this is the same answer you get when evaluating the corresponding infix expression:
(5 + 6 * 5 / (18 – 15)) / 2).

Example 2

Evaluate the following postfix expression: 2 3 2 3 ^ ^ + 6 15 3 - * - 4 +

Next Symbol	**Stack**
2	2
3	2 3
2	2 3 2
3	2 3 2 3
^	2 3 8
^	2 6561
+	6563
6	6563 6
15	6563 6 15
3	6563 6 15 3
-	6563 6 12
*	6563 72
-	6491
4	6491 4
+	6495

Answer: 6495

Verify that this is the same answer you get when evaluating the corresponding infix expression:
(2 + 3 ^ 2 ^ 3 - 6 * (15 - 3) + 4).